**Spring Annotations**

In most typical applications, we have distinct layers like data access, presentation, service, business, etc.

And, in each layer, we have various beans. Simply put, to detect them automatically, Spring uses classpath scanning annotations.

then, it registers each bean in the ApplicationContext.

Here's a quick overview of a few of these annotations:

@Component is a generic stereotype for any Spring-managed component

@Service annotates classes at the service layer

@Repository annotates classes at the persistence layer, which will act as a database repository

There're several ways to configure beans in a Spring container. We can declare them using XML configuration. We can declare beans using the @Bean annotation in a configuration class.

Or we can mark the class with one of the annotations from the org.springframework.stereotype package and leave the rest to component scanning.

Component Scanning

Spring can automatically scan a package for beans if component scanning is enabled.

@ComponentScan configures which packages to scan for classes with annotation configuration. We can specify the base package names directly with one of the basePackages or value arguments (value is an alias for basePackages):

@Configuration

@ComponentScan(basePackages = "com.baeldung.annotations")

class VehicleFactoryConfig {}

Also, we can point to classes in the base packages with the basePackageClasses argument:

@Configuration

@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)

class VehicleFactoryConfig {}

Both arguments are arrays so that we can provide multiple packages for each.

If no argument is specified, the scanning happens from the same package where the @ComponentScan annotated class is present.

@ComponentScan leverages the Java 8 repeating annotations feature, which means we can mark a class with it multiple times:

@Configuration

@ComponentScan(basePackages = "com.baeldung.annotations")

@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)

class VehicleFactoryConfig {}

Alternatively, we can use @ComponentScans to specify multiple @ComponentScan configurations:

@Configuration

@ComponentScans({

@ComponentScan(basePackages = "com.baeldung.annotations"),

@ComponentScan(basePackageClasses = VehicleFactoryConfig.class)

})

class VehicleFactoryConfig {}

When using XML configuration, the configuring component scanning is just as easy:

<context:component-scan base-package="com.baeldung" />

3. @Component

@Component is a class level annotation. During the component scan, Spring Framework automatically detects classes annotated with @Component.

***Or*** *@Component* is an annotation that allows Spring to automatically detect our custom beans.

In other words, without having to write any explicit code, Spring will:

* Scan our application for classes annotated with *@Component*
* Instantiate them and inject any specified dependencies into them
* Inject them wherever needed

**We can use @Component across the application to mark the beans as Spring's managed components**. Spring only pick up and registers beans with @Component  and doesn't look for @Service and @Repository in general.

They are registered in ApplicationContext because they themselves are annotated with @Component:

@Component

**public** @interface Service {

}

@Component

**public** @interface Repository {

}

@Service and @Repository are special cases of @Component. They are technically the same but we use them for the different purposes.

For example:

@Component

class CarUtility {

// ...

}

By default, the bean instances of this class have the same name as the class name with a lowercase initial. On top of that, we can specify a different name using the optional value argument of this annotation.

Since @Repository, @Service, @Configuration, and @Controller are all meta-annotations of @Component, they share the same bean naming behavior. Also, Spring automatically picks them up during the component scanning process.

4. @Repository

DAO or Repository classes usually represent the database access layer in an application, and should be annotated with @Repository:

@Repository

class VehicleRepository {

// ...

}

One advantage of using this annotation is that it has automatic persistence exception translation enabled. When using a persistence framework such as Hibernate, native exceptions thrown within classes annotated with @Repository will be automatically translated into subclasses of Spring's DataAccessExeption.

To enable exception translation, we need to declare our own PersistenceExceptionTranslationPostProcessor bean:

@Bean

public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {

return new PersistenceExceptionTranslationPostProcessor();

}

Note, that in most cases, Spring does the step above automatically.

Or, via XML configuration:

<bean class=

"org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

**@Repository’s job is to catch persistence specific exceptions and rethrow them as one of Spring’s unified unchecked exception**.

5. @Service

The business logic of an application usually resides within the service layer – so we'll use the @Service annotation to indicate that a class belongs to that layer:

@Service

public class VehicleService {

// ...

}

6. @Controller

@Controller is a class level annotation which tells the Spring Framework that this class serves as a controller in Spring MVC:

@Controller

public class VehicleController {

// ...

}

7. @Configuration

Configuration classes can contain bean definition methods annotated with @Bean:

@Configuration

class VehicleFactoryConfig {

@Bean

Engine engine() {

return new Engine();

}

}

## ****What's Different?****

**The major difference between these stereotypes is they are used for different classification.** When we annotate a class for auto-detection, then we should use the respective stereotype

**Spring @GetMapping Example**

@GetMapping is specialized version of @RequestMapping annotation that acts as a shortcut for

@RequestMapping(method = RequestMethod.GET). @GetMapping annotated methods handle the HTTP GET requests matched with given URI expression

@RequestMapping(method = RequestMethod.POST). @PostMapping

**@RequestBody**

the @RequestBody annotation maps the HttpRequest body to a transfer or domain object, enabling automatic deserialization of the inbound HttpRequest body onto a Java object.

First, let's have a look at a Spring controller method:

@PostMapping("/request")

public ResponseEntity postController(

@RequestBody LoginForm loginForm) {

exampleService.fakeAuthenticate(loginForm);

return ResponseEntity.ok(HttpStatus.OK);

}

Spring automatically deserializes the JSON into a Java type, assuming an appropriate one is specified.

**@ResponseBody**

The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

Suppose we have a custom Response object:

public class ResponseTransfer {

private String text;

// standard getters/setters

}

Next, the associated controller can be implemented:

@Controller

@RequestMapping("/post")

public class ExamplePostController {

@Autowired

ExampleService exampleService;

@PostMapping("/response")

@ResponseBody

public ResponseTransfer postResponseController(

@RequestBody LoginForm loginForm) {

return new ResponseTransfer("Thanks For Posting!!!");

}

}

In the developer console of our browser or using a tool like Postman, we can see the following response:

{"text":"Thanks For Posting!!!"}

Remember, we don't need to annotate the @RestController-annotated controllers with the @ResponseBody annotation since Spring does it by default.

Spring MVC 4 - @SessionAttributes or @SessionAttribute

to use the @SessionAttributes and @SessionAttribute annotations in Spring MVC with the help of a small login application.

@SessionAttributes annotation is used to store the model attribute in the session. This annotation is used at controller class level.

@SessionAttributes("user")

public class LoginController {

@ModelAttribute("user")

public User setUpUserForm() {

return new User();

}

}

In the above code snippet, the model attribute ‘user’ will be added to the session if the name attribute of the @ModelAttribute and @SessionAttributes annotations is same.

@SessionAttribute annotation is used to retrieve the existing attribute from session that is managed globally and it is used at method parameter as shown follows.

@GetMapping("/info")

public String userInfo(@SessionAttribute("user") User user) {

//...

//...

return "user";

}

SpringMVCValidationAPI

@NotNull

@Email

@Size

@NotEmpty

In spring MVC , we can validate the form in many ways , we can use validation API ,custom validator or directly validating in controller after the form submit.

Lets see how we can do the same using Java Validation API.

Java Validation API suggests us to use annotation on the model class for each attribute to specify the validation constraints.

Example:

Copy this code

public class Customer {

@NotNull

@Email

private String emailId;

@NotNull

@Size(min=8,max=15)

private String password;

//getters and setters

}

So we are using @NotNull to specify that emailId and password fields should not be null.

@Email – specifies that , emailId field is getting validated as per the email standard provided by Hibernate validator.

@Size specifies that password field should be minimum of 8 characters and maximum of 15 characters.

We can use many other annotations to validate the fields in the form.

In the controller class, specify the form backing model object with @Valid annotation to ensure that form values gets binded with this model object and do the validation.

@Controller

public class LoginController {

@RequestMapping(value = "/doLogin", method = RequestMethod.POST)

public String doLogin(@Valid User user, BindingResult result) {

// login logic here

}

}

If any validation error occurs as per our validation defined in the model class, then BindingResult will have the errors.

if (result.hasErrors()) {

// form validation error, return the same form

} else {

// form input is fine, continue the flow

}

so we can check the binding results for any errors.

If it has errors we can return the same form with error message, if not we can continue the flow.

We can display the error messages on the jsp by using spring’s form errors tag as follows

<form:errors path="emailId" />

Where emailId is the exact attribute name in the Customer Model class.

We can also specify the error message while specifying the validation constraints in the model class with annotation as below

@NotEmpty(message = "Email Id is Required.")

private String emailId;

since error message has to be localized, we will specify it in the properties file.

The way of specifying the key in the properties file is

ConstraintName.modelAttributeName.propertyName=validation error message

Example

NotEmpty.customer.emailId=Email Id is Required.

Spring MVC form validation with custom validator

In the previous example, we have seen the form validation using java validations by annotating model class with constraint validation annotations.

Good part of it is, we can just add annotations on the fields of the model class and if any errors in the form as per the annotations , Binding Result will get the errors.

But bad part of it is , we will not get annotations for all our business validations on the form fields.

Example : If we want to Validate age field to be greater than 18 and less than 60 to be allowed then we need to define our own validator.

So lets see how we can write our custom validator

We need to do following things to define our custom validator

1)Create the custom validator class for model class on which we validate and implement the validator interface, make custom validator class as spring bean.

2)Override supports(Class clazz) method

3)Override validate(Object target,Errors errors) method

4)After this we need to inject this custom validator in the controller class and call its validate() method

5)Then check the binding result for any errors and return the appropriate view.

//==================================================================

Various View Resolver

MultipartResolver: Interface to handle the file uploads

LocaleResolver: Helps to resolve the locale from the request

ThemeResolver: Resolve a theme for a request(CSS)

HandlerMapping: Maps the Request to Handlers (Controllers)

HandlerAdapter: Plugs the other frameworks handlers

HandlerExceptionResolver: Mapping of the exceptions to handlers and views

ViewResolver: Maps the view names to view instances

For Spring Handler Mapping:

https://www.baeldung.com/spring-handler-mappings

HandlerMapping:

1.Bean Name Url Handler Mapping

The BeanNameUrlHandlerMapping class is the default handler mapping class, which maps the URL request(s) to the name of the beans mentioned in the configuration.

<beans>

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/"/>

<property name = "suffix" value = ".jsp"/>

</bean>

<bean class = "org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean name = "/helloWorld.htm"

class = "com.sb.HelloController" />

<bean name = "/hello\*"

class = "com.sb.HelloController" />

<bean name = "/welcome.htm"

class = "com.sb.WelcomeController"/>

</beans>

For example, using the above configuration, if URI

• /helloWorld.htm or /hello{any letter}.htm is requested, DispatcherServlet will forward the request to the HelloController.

• /welcome.htm is requested, DispatcherServlet will forward the request to the WelcomeController.

• /welcome1.htm is requested, DispatcherServlet will not find any controller and server will throw 404 status error.

ControllerClassNameHandlerMapping

The Controller Class Name Handler Mapping using the Spring Web MVC framework. The ControllerClassNameHandlerMapping class is the convention-based handler mapping class, which maps the URL request(s) to the name of the controllers mentioned in the configuration. This class takes the Controller names and converts them to lower case with a leading "/".

For example − HelloController maps to "/hello\*" URL.

<beans>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/"/>

<property name = "suffix" value = ".jsp"/>

</bean>

<bean class = "org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<bean class = "com.sb.HelloController" />

<bean class = "com.sb.WelcomeController"/>

</beans>

For example, using above configuration, if URI

• /helloWorld.htm or /hello{any letter}.htm is requested, DispatcherServlet will forward the request to the HelloController.

• /welcome.htm is requested, DispatcherServlet will forward the request to the WelcomeController.

/Welcome.htm is requested where W is capital cased, DispatcherServlet will not find any controller and the server will throw 404 status error.

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.web.servlet.mvc.AbstractController;

public class HelloController extends AbstractController{

@Override

protected ModelAndView handleRequestInternal(HttpServletRequest request,

HttpServletResponse response) throws Exception {

ModelAndView model = new ModelAndView("hello");

model.addObject("message", "Hello World!");

return model;

}

}

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.web.servlet.mvc.AbstractController;

public class WelcomeController extends AbstractController{

@Override

protected ModelAndView handleRequestInternal(HttpServletRequest request,

HttpServletResponse response) throws Exception {

ModelAndView model = new ModelAndView("welcome");

model.addObject("message", "Welcome!");

return model;

}

}

Spring MVC - Controller Class Name Handler

The ControllerClassNameHandlerMapping class is the convention-based handler mapping class, which maps the URL request(s) to the name of the controllers mentioned in the configuration. This class takes the Controller names and converts them to lower case with a leading "/".

For example − HelloController maps to "/hello\*" URL.

<beans>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/"/>

<property name = "suffix" value = ".jsp"/>

</bean>

<bean class = "org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<bean class = "com.sbtutorialspoint.HelloController" />

<bean class = "com.sb.WelcomeController"/>

</beans>

For example, using above configuration, if URI

• /helloWorld.htm or /hello{any letter}.htm is requested, DispatcherServlet will forward the request to the HelloController.

• /welcome.htm is requested, DispatcherServlet will forward the request to the WelcomeController.

• /Welcome.htm is requested where W is capital cased, DispatcherServlet will not find any controller and the server will throw 404 status error.

Spring MVC XmlViewResolver Configuration Example

In Spring MVC based application, the last step of request processing is to return the logical view name. Here DispatcherServlet has to delegate control to a view template so the information is rendered. This view template decides that which view should be rendered based on returned logical view name. These view templates are one or more view resolver beans declared in the web application context. These beans have to implement the ViewResolver interface for DispatcherServlet to auto-detect them. Spring MVC comes with several ViewResolver implementations. In this example, we will look at such a view resolver template i.e. XmlViewResolver.

Contrary to InternalResourceViewResolver where each logical view name is mapped to physical location of view directly, in case of XmlViewResolver, views are declared as Spring beans. You can declare the view beans in the same configuration file as the web application context, but it’s better to isolate them in a separate configuration file.

By default, XmlViewResolver loads view beans from /WEB-INF/views.xml, but this location can be overridden through the location property.

<bean class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="location">

<value>/WEB-INF/admin-views.xml</value>

</property>

</bean>

In the dispatcher-servlet.xml configuration file, you can declare each view as a normal Spring bean by setting the class name and properties. e.g.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="home" class="org.springframework.web.servlet.view.JstlView">

<property name="url" value="/WEB-INF/jsp/home.jsp" />

</bean>

<bean id="admin/home" class="org.springframework.web.servlet.view.JstlView">

<property name="url" value="/WEB-INF/jsp/admin/home.jsp" />

</bean>

<bean id="logOffRedirect" class="org.springframework.web.servlet.view.RedirectView">

<property name="url" value="home" />

</bean>

</beans>

The first two beans in above configuration are pretty obvious. Logical view name “home” is mapped to “/WEB-INF/jsp/home.jsp” and “admin/home” is mapped to “/WEB-INF/jsp/admin/home.jsp“.

The third bean do not map any physical view file, rather it redirect the request to url “home” which is actually handled by controller of URL “/home“. Whatever logical name that controller will return, that view will be looked up into bean mappings and then actual view file will be obtained.

//==============================

@PathVariable annotation

In Spring MVC, the @PathVariable annotation binds a URI template variable to a method parameter in a controller.

A URI template contains one or more path variables enclosed by braces ({}) as follows.

http://www.example.com/user/{firstName}/{lastName}

When path variables of a URI template are substituted with actual value then it becomes a URI.

http://www.example.com/user/Sunil/Singh

Controller class

Create a @Controller class whose handler methods are annotated with @GetMapping annotation and method parameters are annotated with @PathVariable annotation.

Use the @ResponseBody annotation to bind the handler method’s return type to the web response body.

Spring configuration

Create a web @Configuration class annotated with @EnableWebMvc and @ComponentScan

Servlet container initialization

Create a container initializer class by extending the AbstractAnnotationConfigDispatcherServletInitializer class

The AbstractAnnotationConfigDispatcherServletInitializer class, implements the WebApplicationInitializer, is implemented in Servlet 3.0+ environments in order to configure the ServletContext programmatically.

/==================================

@SessionAttribute== argument level /@SessionAttributes===class level